APPENDIX A – SAMPLE CODE

```
import numpy as np
import pandas as pd
import os
import cv2
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import albumentations as A
import segmentation_models_pytorch as smp
import segmentation_models_pytorch.utils as smpUtils
import matplotlib.pyplot as plt
from tqdm import tqdm
import random

# ================== CONFIGURATION ==================
CSV_PATH = "/Users/jathin/Downloads/project/CrackDetection-main/project1.csv"
MASK_DIR = "/Users/jathin/Downloads/project/CrackDetection-main/masks"
OUTPUT_CSV = "/Users/jathin/Downloads/project/CrackDetection-main/trail.csv"
DEVICE = 'mps' if torch.backends.mps.is_available() else 'cpu'
ENCODER = "resnet34"
ENCODER_WEIGHTS = "imagenet"
BATCH_SIZE = 4
EPOCHS = 45
MODEL_PATH = "/Users/jathin/Downloads/project/CrackDetection-
main/best_model.pth"
```

```
# Create masks directory if it doesn't exist
os.makedirs(MASK_DIR, exist_ok=True)

# ================= MASK GENERATION =================
def generate_masks():
    """Generate binary masks for all images in the CSV file"""
    print("Generating masks for crack detection...")

    # Load dataset CSV
    if not os.path.exists(CSV_PATH):
        raise FileNotFoundError(f"CSV file not found: {CSV_PATH}")

    df = pd.read_csv(CSV_PATH)
    print(f"Loaded {len(df)} images from CSV")

    # Create mask column in dataframe
    df['mask_path'] = None

    # Process each image to create mask
    for idx, row in tqdm(df.iterrows(), total=len(df)):
        img_path = row['image_path']
        if not os.path.exists(img_path):
            print(f"Warning: Image not found: {img_path}")
            continue

        # Extract filename without extension
        img_name = os.path.basename(img_path)
        file_name, ext = os.path.splitext(img_name)
```

```
        # Define mask path
        mask_path = os.path.join(MASK_DIR, f"{file_name}_mask.png")
        df.at[idx, 'mask_path'] = mask_path

        # Skip if mask already exists
        if os.path.exists(mask_path):
            continue

        # Create mask (adjust parameters for your specific crack types)
        try:
            # Read image
            image = cv2.imread(img_path)
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

            # Apply Gaussian blur to reduce noise
            blur = cv2.GaussianBlur(gray, (5, 5), 0)

            # Apply adaptive thresholding (good for cracks with varying lighting)
            thresh = cv2.adaptiveThreshold(
                blur, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                cv2.THRESH_BINARY_INV, 25, 3
            )

            # Clean up noise with morphological operations
            kernel = np.ones((3, 3), np.uint8)
            opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel,
iterations=1)
```

```
        # Remove small connected components
        nb_components, output, stats, centroids =
cv2.connectedComponentsWithStats(opening, connectivity=8)
        sizes = stats[1:, -1]
        min_size = 50  # Minimum size of crack components

        # Create clean mask
        mask = np.zeros_like(output)
        for i in range(1, nb_components):
            if sizes[i-1] >= min_size:
                mask[output == i] = 255

        # Save mask
        cv2.imwrite(mask_path, mask)

        # Visualize first few masks
        if idx < 5:
            plt.figure(figsize=(12, 4))
            plt.subplot(1, 3, 1)
            plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
            plt.title("Original Image")
            plt.axis('off')

            plt.subplot(1, 3, 2)
            plt.imshow(mask, cmap='gray')
            plt.title("Generated Mask")
            plt.axis('off')
```

```
            # Overlay for visualization
            overlay = cv2.cvtColor(image.copy(), cv2.COLOR_BGR2RGB)
            overlay[mask > 0] = [255, 255, 0]  # Yellow highlight for cracks

            plt.subplot(1, 3, 3)
            plt.imshow(overlay)
            plt.title("Overlay")
            plt.axis('off')
            plt.tight_layout()
            plt.show()

        except Exception as e:
            print(f"Error processing {img_path}: {e}")

    # Save the updated CSV with mask paths
    df.to_csv(OUTPUT_CSV, index=False)
    print(f"Generated masks and saved CSV to {OUTPUT_CSV}")
    return df

# ================== DATASET CLASS ==================
class CrackDataset(Dataset):
    def __init__(self, df, transform=None):
        self.df = df
        self.transform = transform

    def __len__(self):
        return len(self.df)
```

```
    def __getitem__(self, idx):
        img_path = self.df.iloc[idx]['image_path']
        mask_path = self.df.iloc[idx]['mask_path']

        # Load image and mask
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
        # Binarize mask if needed
        _, mask = cv2.threshold(mask, 127, 1, cv2.THRESH_BINARY)

        # Apply transformations
        if self.transform:
            transformed = self.transform(image=image, mask=mask)
            image = transformed['image']
            mask = transformed['mask']

        # Convert to tensors
        image = torch.from_numpy(image).permute(2, 0, 1).float() / 255.0
        mask = torch.from_numpy(mask).long().unsqueeze(0)

        return image, mask

# ================== TRANSFORMS ==================
def get_transforms():
    train_transform = A.Compose([
```

```
        A.Resize(256, 256),
        A.HorizontalFlip(p=0.5),
        A.ShiftScaleRotate(scale_limit=0.5, rotate_limit=0, shift_limit=0.1, p=0.5, border_mode=0),
        A.GaussNoise(p=0.2),
        A.RandomBrightnessContrast(p=0.5),
    ])

    val_transform = A.Compose([
        A.Resize(256, 256),
    ])

    return train_transform, val_transform

# ================== TRAINING LOOP ==================
def train_model(df):
    """Train the crack detection model"""
    print(f"\nTraining crack detection model on {DEVICE}...")

    # Prepare transforms
    train_transform, val_transform = get_transforms()

    # Split data
    train_df = df.sample(frac=0.8, random_state=42)
    val_df = df.drop(train_df.index)

    # Create datasets
    train_dataset = CrackDataset(train_df, transform=train_transform)
```

```
val_dataset = CrackDataset(val_df, transform=val_transform)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)

# Create model
model = smp.UnetPlusPlus(
    encoder_name=ENCODER,
    encoder_weights=ENCODER_WEIGHTS,
    in_channels=3,
    classes=2,
    activation='sigmoid'
).to(DEVICE)

# Define loss and metrics
loss_fn = smpUtils.losses.DiceLoss()
metrics = [
    smpUtils.metrics.Fscore(threshold=0.5),
    smpUtils.metrics.Accuracy(threshold=0.5)
]

# Define optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training epochs
train_epoch = smpUtils.train.TrainEpoch(
    model,
```

```
    loss=loss_fn,
    metrics=metrics,
    optimizer=optimizer,
    device=DEVICE,
    verbose=True
)

val_epoch = smpUtils.train.ValidEpoch(
    model,
    loss=loss_fn,
    metrics=metrics,
    device=DEVICE,
    verbose=True
)

# Train model
best_score = 0
for epoch in range(1, EPOCHS + 1):
    print(f"\nEpoch {epoch}/{EPOCHS}")

    # Train
    train_logs = train_epoch.run(train_loader)

    # Validate
    val_logs = val_epoch.run(val_loader)

    # Save best model
    if val_logs['fscore'] > best_score:
```

```
            best_score = val_logs['fscore']
            torch.save(model.state_dict(), MODEL_PATH)
            print(f"Model saved with F-score: {best_score:.4f}")

        # Reduce learning rate after half the epochs
        if epoch == EPOCHS // 2:
            optimizer.param_groups[0]['lr'] = 0.0001
            print("Reduced learning rate to 0.0001")

    print("Training completed!")
    return model

# ================== VISUALIZATION ==================
def visualize_predictions(df, num_samples=5):
    """Visualize predictions from the trained model"""
    print("\nVisualizing crack detection results...")

    # Load model
    model = smp.UnetPlusPlus(
        encoder_name=ENCODER,
        encoder_weights=ENCODER_WEIGHTS,
        in_channels=3,
        classes=2,
        activation='sigmoid'
    ).to(DEVICE)

    if os.path.exists(MODEL_PATH):
        model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
```

```
    print(f"Loaded model from {MODEL_PATH}")
else:
    print(f"Warning: Model not found at {MODEL_PATH}")
    return

# Set up transform
_, val_transform = get_transforms()

# Create dataset
dataset = CrackDataset(df, transform=val_transform)

# Visualize random samples
model.eval()
indices = random.sample(range(len(dataset)), min(num_samples, len(dataset)))

with torch.no_grad():
    for idx in indices:
        # Get image and mask
        image, gt_mask = dataset[idx]

        # Get prediction
        x_tensor = image.to(DEVICE).unsqueeze(0)
        pred_mask = model(x_tensor)

        # Process prediction
        pred_probs = nn.Softmax(dim=1)(pred_mask)
        pred_mask = torch.argmax(pred_probs, dim=1).squeeze().cpu().numpy()
```

```
        # Convert for visualization
        image_vis = image.permute(1, 2, 0).cpu().numpy()
        gt_mask_vis = gt_mask.squeeze().cpu().numpy()

        # Create plot like in your example
        plt.figure(figsize=(15, 5))

        plt.subplot(1, 3, 1)
        plt.imshow(image_vis)
        plt.title("Image")
        plt.axis('off')

        plt.subplot(1, 3, 2)
        plt.imshow(gt_mask_vis, cmap='viridis')
        plt.title("Ground Truth Mask")
        plt.axis('off')

        plt.subplot(1, 3, 3)
        plt.imshow(pred_mask, cmap='viridis')
        plt.title("Predict Mask")
        plt.axis('off')

        plt.tight_layout()
        plt.show()

# ================= MAIN FUNCTION =================
def main():
    # Step 1: Generate masks
```

```
    df = generate_masks()

    # Step 2: Train model
    train_model(df)

    # Step 3: Visualize results
    visualize_predictions(df)

if __name__ == "__main__":
    main()
```